Chapter 20

Struct and Unity examples

This section is more examples with the `Vector3` and `Color` structs, some built-ins and playing with some Unity features that need structs to use.

20.1 More Vector3, Color code

This first thing is a rewrite of the old movement code, to show off `Vector3`'s. It doesn't run any better, but it might look a little nicer.

Since the Unity system thinks position is a `Vector3`, we should store ours in one. Instead of declaring `x`, `y` and `z` for our movement variables, like we did before, we can declare `Vector3 pos;`.

This is the move&wrap code, rewritten with a Vector3 replacing `float x` we had before:

```
Vector3 pos;

void Start() {
  pos.x=-7; pos.y=2; pos.z=0;
  transform.position=pos;
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) pos.x=-7;

  transform.position = pos;
}
```

Notice how we still need to set `transform.position` to really move us. `pos` is still a normal variable, with no meaning until we use it for something.

A neater example of this is rewriting the "each lap is a random y" version. We needed global x and y for that before. Now declaring `Vector3 pos;` gives us both. But otherwise the code is the same:

```
Vector3 pos;

void Start() { // no changes in start
  pos.x=-7; pos.y=2; pos.z=0;
  transform.position=pos;
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) { // wrap-around
    pos.x=-7;
    pos.y = Random.Range(-3.0f, 3.0f); // <-- new line
  }
  transform.position = pos; // copies the x and y we changed
}
```

This next example is a little different. I'd like to cycle through three different colors. To get the colors, I'll declare three `Color` variables. That's the main point of this example – we could do this by declaring 9 floats (rgb for each color), but it's nicer to use 3 Colors, now that we have structs.

We could set them in the program, but we may as well just do it through the Inspector. Here's the first part, with a function to set us to the correct color:

```
public Color c0, c1, c2; // set these using Inspector
int curColNum=0; // 0, 1 or 2. Which Color we're on now
int delay=40;

void Start() {
  setToCurCol();
}

void setToCurCol() {
  Color cc;
  if(curColNum==0) cc=c0;
  else if(curColNum==1) cc=c1;
  else cc=c2;
  GetComponent<Renderer>().material.color=cc;
}
```

`setToCurCol` is one of those sloppy global-using functions for just this program. It uses the compute-then-use idea: the cascading `if` gets the correct color into temporary Color variable `cc`, then the last line uses `cc` to set our color.

The old way, we'd be changing and setting lots of `r`, `g`, `b`'s. I think this way feels nicer – we assign a color to a color.

The Update part isn't special. It just moves `curColNum` through 0,1,2,0 . . . , with a delay:

```
void Update() {
  delay--;
  if(delay<=0) {
    delay=40;
    curColNum++;
    if(curColNum>2) curColNum=0;
    setToCurCol();
  }
}
```

This part doesn't even use colors. It just slow spins a simple int through the 0,1,2 color numbers. But that's part of the point – most of the work is just moving numbers around.

I want to grow the example a little, to show that `Color` variables are still just variables, and everything is under our control. I'll change the last color to be random, so it shows `c0`, `c1` then a random color. I also want it so pressing the space-bar picks new random `c0` and `c1` colors (we'll see it spin through those exact new colors, until we press space again).

This is the whole thing redone, including my random color function from before:

```
public Color c0, c1; // 2 colors in the sequence (may change)
int curColNum=0; // which Color we're on now: 0, 1 or 2
int delay=40;

void Start() {
  setToCurCol();
}

float rand01() { return Random.Range(0,1.0f); }

Color randCol() {
  Color cc; cc.r=rand01(); cc.g=rand01(); cc.b=rand01(); cc.a=1;
  return cc;
}
```

`rand01()` is a typical "helper" function. All it does is make `randCol` shorter and easier to write.

The rest:

```
void setToCurCol() {
  Color cc;
  if(curColNum==0) cc=c0;
  else if(curColNum==1) cc=c1;
  else cc=randCol(); // <- last color is random
  GetComponent<Renderer>().material.color=cc;
}

void Update() { // no change in this part:
  delay--;
  if(delay<=0) {
    delay=40;
    curColNum++;
    if(curColNum>2) curColNum=0;
    setToCurCol();
  }

  // pressing space rerolls the first two colors:
  if(Input.GetKeyDown(KeyCode.Space)) {
    c0=randCol();
    c1=randCol();
  }
}
```

This should cycle through 2 familiar colors then a random one, over and over. Pressing space appears to give random colors, but then we'll see same, same, random after a few repeats. It works since Color variables are still just regular variables, which we can change and play with in the usual ways.

20.2 More built-in functions

Most built-in structs have built-in functions to do useful things with them. This section uses two fun ones made for `Vector3`'s: `Distance` and `MoveTowards`.

They're normal functions, but have one bit of weirdness. They're both in the Vector3 namespace – you find them with `Vector3.Distance` and `Vector3.MoveTowards`. C# likes to double-use struct names this way. Vector3 is a struct, and it's also a namespace holding regular functions.

It's nice that you can find built-in Vector3 functions by typing `Vector3`-dot then reading what it says.

20.2.1 Distance

`Distance` takes two `Vector3` points and tells you how far apart they are:

```
public Vector3 v1, v2;
```

```
  float d = Vector3.Distance(v1, v2);
  if(d<2) print("too close");
```

It's a pure math function. It reads the input points and tells you a number.

Now that we know out position is a Vector3, we can check how far we're away from something:

```
public Vector3 target;

void Update() {
  // pretend code moves us

  if( Vector3.Distance(transform.position, target) < 0.2f )
    print("we're at the target location");
}
```

20.2.2 MoveTowards

`MoveTowards` is named for what people usually do with it – move themselves towards some target. It's useful because it will even move the correct distance diagonally.

The official heading looks like this:

```
Vector3 MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta)
```

Remember, the parameter names are just made up: `current` and `target` are good descriptive names, but don't let `maxDistanceDelta` scare you. It's a float for how far we want to move.

`Vector3.MoveTowards(p1, p2, 0.1f);` says to pretend you're at `p1` and you moved 0.1 towards `p2`. Where would you be? It's another pure math function – it doesn't actually move or change anything – merely returns the spot.

The most common use is: `pos = MoveTowards(pos, target, 0.1f);`. You may recognize that form, it's the same idea as `n=abs(n);`. It changes `pos` because we specially added an `=` to make it change `pos`.

MoveTowards has one extra thing it does: it won't overshoot.
`MoveTowards(p1,p2,10);` will move 10 units from `p1` to `p2`, only if they're 10 or more apart. Otherwise it stops at `p2`.

Altogether this code will creep `pos` towards `target` and then stop:

```
public Vector3 pos, target;

void Update() {
  pos = Vector3.MoveTowards(pos, target, 0.1f);
}
```

When it gets to target, the program keeps running. MoveTowards runs every Update, changing `pos` to the new value. But the new value is always the same. It's a neat trick.

20.2.3 More movement examples

Here's a short program using MoveTowards and Distance. It moves the Cube from a random spot, to the center of the screen, over and over. The first line does the movement. The `if` is for resetting. I'm assuming the Camera is in a front-view:

```
public Vector3 pos, targ; // targ starts at 000

void Update() {
  pos = Vector3.MoveTowards(pos, targ, 0.1f);

  // random teleport when we get close enough:
  float dist=Vector3.Distance(pos, targ);
  if(dist<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);
  }
  transform.position=pos;
}
```

That should move directly to the center, pop away, and repeat. Sometimes it might pop next to the center and almost immediately pop away again, but that's just how random works.

The program works for any target position. We could manually put `targ` in a corner. But it would be cool to have the program change the target as it runs. This adds our old lap-counter to randomize the target every 5 laps:

```
public Vector3 pos, targ;
int laps=0;

void Update() {
  pos = Vector3.MoveTowards(pos, targ, 0.1f);

  // random teleport when we get there:
  if(Vector3.Distance(pos, targ)<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);

    // change target every 5 trips:
    laps++;
    if(laps>=5) {
      laps=0;
      targ.x = Random.Range(-7.0f, 7.0f);
      targ.y = Random.Range(-5.0f, 5.0f);
    }
  }
  transform.position=pos;
}
```

The moral here is that we can find some cool functions we didn't know about, like diagonal movement towards a target. But using it in a program is usually the same old tricks.

This version speeds up towards the target. We change the 0.1 movement into a variable. It starts at 0 after each teleport, and increases as we move:

```
public Vector3 pos, targ; // targ starts at 000
float mvSpeed=0;

void Update() {
  pos = Vector3.MoveTowards(pos, targ, mvSpeed);
  mvSpeed += 0.01f; // check: will reach 0.1 in 10 Updates

  float dist=Vector3.Distance(pos, targ);
  if(dist<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);
    mvSpeed=0; // after a pop, needs to speed up from 0
  }
  transform.position=pos;
}
```

20.2.4 Color namespace

`Color` is also re-used as a namespace, holding preset global color variables:

```
public Color c1, c2;

void Start() {
  c1 = Color.red; // <- a pre-made global
  c2 = Color.green;
}
```

The trick is that we're double-using Color. It means 2 different things. As usual `Color c1, c2;` declares variables. But `Color.red` isn't declaring anything. It's how we find a pre-set global variable. It's a shortcut for `new Color(1,0,0)`.

Color also has a function hidden in it. Lerp for colors lets you find a color in-between 2 others. This gives a color halfway between c1 and c2:

```
public Color c1, c2;
public Color cHalf;

void Start() {
  // cHalf is an average of c1 and c2:
  cHalf = Color.Lerp(c1, c2, 0.5f);
}
```

When you see `Color.Lerp` you just have to know we're not declaring a variable. It's a normal function named `Lerp` in the Color namespace.

20.3 Transform struct

After using `transform.position` and `transform.localScale`, you may have realized that transform is a struct. When you type `transform`-dot and get a pop-up, that's the normal struct menu showing the possible fields. The Transform struct looks pretty much like this:

```
public struct Transform {
  public Vector3 position;
  public Vector3 localScale;
  public string name; // <-new
  ... // <- more stuff we don't care about
}
```

Unity automatically declares a global `transform`, of type `Transform`. That's what we've been using this whole time to move around.

`transform` is basically a regular variable. The only thing magic is that the system is always checking it and changing the display based on the values.

We can write a fun program changing the name. It's pretty much the same thing as `c1.name="Bessy"`, except that the system reads the new name and displays it in the object panel:

```
public string name="I stay the same";

void Start() {
  print("Old name: " + transform.name);
  transform.name = "abc" + Random.Range(1,1000);
  // look in Inspector/Hierarchy - name has changed
}
```

I declared `string name;` to show off the local-only rule for field names. Transforms have `name`, but anyone else can have one, too, plus we can declare our own `name` variable. The system will never confuse them.

`transform.position.x` should now make sense. It's a struct in a struct. `transform.position` is a Vector3, which means you use dot-x, dot-y or dot-z. Typing `transform.x` is an error for the normal reason – you can't jump past a field. You have to say dot-position or dot-localScale.

20.3.1 Special errors; pull-out trick

Arrrrg! `transform.position.x=0;` is an error! The rules say it should be fine, but Unity needed some extra tricks to run correctly and messed it up. You get the error *Cannot modify a value type return value of 'UnityEngine.Transform.position'. Consider storing the value in a temporary variable.* Arrg!.

But that's fine. The work-around is also a good way to program. Instead of reaching inside of something, it's often good to "pull it out", make changes, then copy it back. Here's how to change the position:

```
  Vector3 p = transform.position; // copy the whole position
  p.x+=0.1f; // playing with a Vector3 is fine
  if(p.y>5) p.y=5; // here, too
  transform.position = p; // copy it back
```

This next program does the same thing with the scale. It shrinks the x-scale down to 0.5 while leaving the rest the same. It's not super exciting, but you can reset the scale as it runs, and it will go down again:

```
void Update() {
  Vector3 p = transform.localScale; // copy of the scale
  if(p.x>0.5f) {
    p.x-=0.002f;
    transform.localScale = p; // copy it back
  }
}
```

20.4 Rigidbodies

Most game engines have something they call *physics*. It means you can set up an object to automatically act like a real ball. You can push it once, and it rolls, falls, bounces, and eventually stops. It does all of that automatically, with no script needed.

The reason we care is that we can jump in with programming to change it: the speed is a `Vector3`, and there's a standard programming way to check when we hit something.

Some fun background: the whole name is rigidbody physics. Real objects bend and flex a little. Some bend a lot. Assuming they're all completely solid and stiff – rigid – makes the math easier.

20.4.1 Simple physics set-up

To start, let's get a Cube that falls and bounces around. We'll need a Cube with no script (can remove the script, or make a fresh cube) and a front view Camera (the one showing x and y movement), since gravity makes you fall on `y`. We'll also eventually need a floor and walls (or it won't stay on the screen).

To let the Unity system know it should auto-move the Cube, select it, find `Component` on the top bar, and select `Component->Physics->Rigidbody`. The Cube's Inspector should now have a mini-panel named Rigidbody. Just in case, make sure the ball is somewhere the camera can see it, maybe a little near the top, like (0,4,0), and Play. It should fall, hit the floor, and stop.

To see it bounce more, we can tilt the Cube so it doesn't hit flat. Giving it a z-rotation of 10-30 degrees should make it fall and bounce sideways (just type 20 into the z part of Rotation in the Inspector panel). It should rock just a little, then stop.

The system has a way to set an object's slipperiness and bounciness. As you can see, the default setting is like a bean-bag – not bouncy at all. To give ourselves more to work with, we can make it very bouncy. It takes two steps. We have to make "Bouncy," then apply it to the Cube.

Down in the Assets panel, select `Create->PhysicMaterial`. That should create something with a green picture of a Bounce, named `NewPhysicMaterial`, asking you to rename it. If you like, name it Bouncy (but the name won't matter).

Select the new PhysicMaterial and look at its Inspector. The Bounciness setting is a 0-1 percent for how much it will bounce back. Set `Bounciness` to around 0.9 or so. Then set the `BounceCombine` dropdown to Maximum.

We can use that to make anything be bouncy. Drag that PhysicMaterial onto the Cube. Play should now have the cube really bounce and roll around for a while. It will fall off the edge unless you put some walls there (back in the section about setting up a nicer Scene). Just in case, the actual location of the PhysicMaterial is in the Cube, under BoxCollider (pop it open) in the Material slot (starts with `[none(Physic Material)]`). It will have your new PhysicMaterial if it worked.

That's a lot. I'll sum up, but this stuff is also easy enough to look up, if you know the terms:

- Add a Rigidbody component to a Cube, with no scripts. Play should have it fall.
- Add at least a floor, if you don't have one (a wide, deep, short Cube), so we can see it bounce. Rotate the Cube slightly, so the bottom is tilted and it bounces to one side.
- Create a PhysicMaterial. Set bounciness=0.9; BounceCombine=Maximum. Drag the PhysicMaterial onto the Cube. It should now bounce more.

Plus, if you have trouble with this, there are lots of Unity places to read about these, and what some of the other settings do.

20.4.2 Playing with velocity

The physics system stores our current speed in a `Vector3`. We can't see it in the Inspector, but we can in code. It's `GetComponent<Rigidbody>().velocity;`. This code would fling us sideways:

```
void Start() {
  Vector3 spd = new Vector3(10,0,0); // speed is 10 going right
  GetComponent<RigidBody>().velocity=vel;
}
```

It feels funny since we can set speed once, then let it move by itself. The speed is in units per second, which is why 10 is a good number.

The other odd thing is how the system is constantly changing it. Each Update it applies gravity – it subtracts a tiny bit from `y` the same as we previously did by hand. When the cube hits something, `velocity` is flipped. That's how it makes the bounces.

Normal real-world gravity is about 10 meters a second, each second. If `velocity.y` is 10, it will go up, slow down, and start to fall after 1 second. This will launch us in a high arc, coming back down after about 2 seconds (during which time it will have gone 4 units to the right):

```
void Start() {
  Vector3 vel;
  vel.x=2; vel.y=10; vel.z=0; // strong push up, a little to the right
  GetComponent<Rigidbody>().velocity = vel;
}
```

We can make that a little nicer by having the space bar give us a random extra pop. Pressing space replaces the old `velocity`. We magically change direction:

```
void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    beginRandomPopUp();
  }
}

void beginRandomPopUp() {
  Vector3 vel;
  vel.x=Random.Range(-5.0f, 5.0f); // random left/right
  vel.y=Random.Range(5.0f, 15.0f); // up 5-15
  vel.z=0; // z is towards/away. Don't let it change
  GetComponent<Rigidbody>().velocity = vel;
}
```

A fun effect is to give us a fake bouncy floor. In this next code, dropping below y=-3, gives the random upward pop (make sure your real floor is below this, so it can fall that far. My floor is down at -5):

```
void Update() {
  if(transform.position.y<-3) {
    beginRandomPopUp();
  }
}
```

This gives completely unrealistic bounces since it ignores your current speed. You could be falling slowly and get a big leftward bounce, or the opposite. A variation is to give an upwards push, using `+=` to add a little each update. This next code will gradually slow us, then push us up, giving an effect like bobbing in water. I'm starting the push when it gets below 0, since it takes a while to reverse direction:

```
void Update() {
  if(transform.position.y<0) {
    Vector3 vel = GetComponent<Rigidbody>().velocity; // pull out
    vel.y+=0.3f; // number is trial&error
    GetComponent<Rigidbody>().velocity=vel; // put back
  }
}
```

The main point here is, sure, the physics system is magical and strange. But once we know that `velocity` is a normal Vector3, and what the units are, we can mess with it the regular way.

20.4.3 Collision callback

The usual term for something that could happen at any time is an **event**. A keypress is a typical event.

There are two basic ways of getting events. One way is a command that you have to write, which checks for it. `if(Input.GetKeyDown(KeyCode.A))` is an example of that. That's offiicially called **polling**. I like that name, because it sounds like what you do – you ask every Update, like you're taking a poll.

The other way is having a callback function. You write a function and register it somehow. When the event happens, your function automatically runs.

In the physics system, colliding with something is an event. You react to it using the second method – write a function which Unity automatically runs. The registration is in the name. `void OnCollisionEnter()` is automatically called when you hit something.

Here's a very short collision callback. It makes you a little smaller after each hit. There's no Start or Update (you can also have them, but don't need them):

```
float sz=1; // current size. 1 is normal

void OnCollisionEnter() {
  sz-=0.1f; if(sz<0.1f) sz=0.1f; // shrink down to 0.1
  transform.localScale = new Vector3(sz,sz,sz);
}
```

The neat thing is how ordinary it is. The body is something we already know how to do. If it was in Update, it would very quickly run 9 times and shrink all the way. Nothing mysterious about it.

As a function, it's also very ordinary. Anyone could call it to make us a bit smaller. The heading looks like nothing special. But like Start and Update, Unity decided to key off of that exact name. When there's a collision, it tries to run our `OnCollisionEnter` function because that's the rule it made.

Here's another one that uses a counter to re-drop us after the 4th bounce. It snaps us to the upper middle, with a random sideways speed:

```
int bounces=0;

void Start() { resetDrop(); }

void OnCollisionEnter() {
  bounces++;
  if(bounces>=4) {
    bounces=0;
    resetDrop();
  }
}

void resetDrop() {
    transform.position = new Vector3(0,5,0); // upper-center
    // toss it a little left or right:
    Vector3 toss; toss.z=0; toss.y=0;
    toss.x=Random.Range(-4.0f, 4.0f);
    GetComponent<Rigidbody>().velocity=toss;
}
```

I used a separate `resetDrop` function mostly to show `OnCollisionEnter` can call other functions. If you think about it, Start and Update are magically called by the system, and they can do anything. So OnCollisionEnter can, too.

20.4.4 The Collision struct

An obvious problem with `OnCollisionEnter()` is it doesn't tell you what you hit. Along with that, it might also be nice to know what part of us hit, and other stuff. There's a better version of `OnCollisionEnter` which tells you that.

To make it look nice, Unity created a struct who's only purpose is to hold collision data, named `Collision`. Here's a partial listing for it:

```
struct Collision {
  public Transform transform; // what we hit
  public Vector3 relativeVelocity; // combined hit speed
  ...
}
```

The idea for making this struct is the usual. To give someone collision data you could send them 10 variables, or you could give them 1 Collision variable holding all 10 things.

The name of the first field, `transform` is a good example of the basic struct rules. In our script, `transform` means us. But `colsn1.transform` is a different variable. As a field of a collision, sent to us, it clearly means the transform of what we hit.

To get collision data, write the collision callback with a `Collision` parameter: `void OnCollisionEnter(Collision col)`. The system checks for your function with and without `col`. If it's there, it sends you collision info. This is very standard. Most callback functions have an input describing details of the event.

A key thing is the system makes and send the Collision. You will never have to create one or even declare one. You never need to look at or understand the collision parts we don't care about. In fact, some of them are rarely used by anyone.

Here's a simple collision callback with an input. It prints the name of whatever we hit:

```
void OnCollisionEnter(Collision col) {
  string myName = transform.name;
  string hitName = col.transform.name;
  print( myName + " ran into " + hitName);
}
```

If your floor and walls have been renamed you'll see things like "Cube hit floor" and "Cube hit Rwall". Not very exciting, but it proves we can check for what we hit, using `col.transform.name`.

This next code checks and tries to climb the walls. It bounces normally off the floor, but hitting anything else shoots it upwards. It assumes the floor is named "floor," and works better if you use Start to give a sideways push:

```
void Start() {} // <-- give it a sideways shove here, to hit walls

void OnCollisionEnter(Collision col) {
  if(col.transform.name != "floor") {
    Vector3 vel = GetComponent<Rigidbody>().velocity; // pull out
    vel.y=Random.Range(5.0f, 10.0f); // snap to an upwards shove
    GetComponent<Rigidbody>().velocity = vel; // put back
  }
}
```

The code only changes the y-speed, leaving the sideways speed alone. When we hit a wall, we bounce off and go the other way, but with a big upwards push. It's a neat effect for such a small, simple amount of code.

We can take a look at other fields inside the collision struct. It has one named `impulse`. Let's print it after each hit:

```
void OnCollisionEnter(Collision C) {
  Vector3 imp = C.impulse; // copy to make the next line shorter
  print( imp.x + ", " + imp.y + ", " + imp.z );
}
```

I still don't know what impulse is, but this is definitely a step to figuring it out.

Notice how I named the input `C` this time. Normally we can name inputs whatever we want. Having OnCollisionEnter called automatically doesn't change that. The system happily fills in the Collision, no matter what we call it.

20.4.5 Misc event notes

In general, any event can be handled either way – polling or a callback – and different systems mix&match. For example, some C# systems handle keys using a callback. It looks a bit like this (I'm making up an example where A and D move us):

```
void OnKeyPress(KeyCode key) { // not legal in Unity
  if(key == KeyCode.A) moveDir=-1;
  else if(key == KeyCode.D) moveDir=+1;
  else moveDir=0;
}
```

When you press `A`, the system would automatically call `OnKeyPress(KeyCode.A)`. Update wouldn't have anything about key presses.

Unity knows about callbacks through the function names. Most other systems have you use a command to register, which also means you can name them whatever you want. For example:

```
// Ex of registering functions. Totally made up and won't run:
```

```
void bounceColorChange(Collision col) { ... }

void Start() {
  // register the bounceColorChange function as a collision enter event:
  collision.enter += bounceColorChange;
}
```

Whenever you want to check for something happening, you have to figure out, is it done with polling, or a callback? If it's with polling, what's the command? If it's a callback, you have to look up how they want you to write the function, and how to register it as the callback.

But then the rest is just programming.